RB-Raft:一种抗拜占庭节点的 Raft 共识算法 *

李淑芝 [1]，邹懿杰 [1]，邓小鸿 [2†]，罗志琼 [1]，刘惠文 [2]

(1. 江西理工大学 信息工程学院，江西 赣州 341000; 2. 赣南科技学院 电子信息工程学院，江西 赣州 341000)

摘　要：针对 Raft 算法无法抵抗拜占庭节点的攻击和日志易窜改等问题，设计了一种抵抗拜占庭节点的 RB-Raft(Resist Byzantine-Raft)算法。首先采用哈希链的方式对每一块日志进行迭代哈希处理，通过动态验证机制对日志进行验证使得对 Leader 节点的恶意行为具有一定的容错率，解决了日志伪造与验证的问题。其次，提出基于门限加密的“遗书”机制，使得 Candidate 节点拉取选票具有合法性，防止拜占庭节点随意拉取选票更换 Leader 节点的攻击，解决了拜占庭节点影响系统一致性的问题。实验结果表明，提出的 RB-Raft 算法具有抗拜占庭节点的能力，其日志识别率可以达到 100%。同时，相比 PBFT，提出的算法共识时延降低了 53.3%，并且吞吐量提高了 61.8%。提出的算法适用于在不可信联盟链中进行共识。

关键词：共识机制；拜占庭容错；哈希链；门限加密；遗书机制

中图分类号：TP393.0　　doi: 10.19734/j.issn.1001-3695.2022.03.0090

Rb-raft:raft consensus algorithm for anti-byzantine nodes

Li Shuzhi[1], Zou Yijie[1], Deng Xiaohong[2†], Luo Zhiqiong[1], Liu Huiwen[2]

(1. *College of Information Science*, *Jiangxi University of Science & Technology*, *Ganzhou Jiangxi* 341000, *China*; 2. *School of Electronics & Information Engineering*, *Gannan University of Science & Technology*, *Ganzhou Jiangxi* 341000, *China*)

Abstract: Aiming at the problems that the Raft algorithm cannot resist the attacks of Byzantine nodes and the logs are easy to tamper with, this paper proposes an RB-Raft (Resist Byzantine-Raft) algorithm that resists Byzantine nodes. Firstly, this paper uses the method of hash chain to iteratively hash each log. At the same time, verification of the log through the dynamic verification mechanism, so that the malicious behavior of the leader node has a certain fault tolerance rate, which solves the problem of log forgery and verification. Secondly, this paper proposes a "Legacy" mechanism based on threshold encryption, which makes it legal for Candidate nodes to pull votes. This mechanism can prevent Byzantine nodes from randomly the attack of pulling votes to replace leader nodes, and solves the problem that Byzantine nodes affect the system consistency. The experimental results show that the proposed RB-Raft algorithm has the ability to resist Byzantine nodes, and its log recognition rate can reach 100%. At the same time, compared with PBFT, the consensus latency of the algorithm in this paper is reduced by 53.3%, and the throughput is increased by 61.8%. The algorithm proposed in this paper is suitable for consensus in untrusted consortium chains.

Key words: consensus mechanisms ;Byzantine fault tolerance; hash chain ; threshold encryption ; “legacy” mechanism

0　引言

中本聪于 2008 年发表的论文《Bitcoin:A Peer-to-Peer Electronic Cash System》[1]标志着比特币的正式推出。在随后的十几年来，区块链作为比特币的底层核心技术不断发展，衍生出了不同类型的区块链，如：以以太坊[2]为代表的公链、以 Hyperledger Fabric[3]为代表的联盟链、以及阿里巴巴的蚂蚁区块链[4]为代表的私链等等。区块链是一种去中心化、不可窜改、可追溯的分布式数据库系统，融合了经济学、P2P 网络、共识算法、非对称加密等多种技术，是互联网技术高度融合的产物。区块链作为一种分布式数据库系统，其最重要的问题就是设计一种共识算法[5,6]来解决分布式节点之间数据的一致性。在不同的区块链中采用的共识算法不同，比特币中采用的是工作量证明[7](Proof of Work，PoW)，而联盟链中一般采用的是实用拜占庭容错[8](Practical Byzantine Fault Tolerance，PBFT)，而在私链中就一般采用经典的一致性算法如：Raft[9]、Paxos[10]。其中 Raft 算法在现有的工程领域应用广泛，是具有强一致性、高性能、高可靠的分布式协议，相比于 Paxos 易于理解和实现，但是因不能够抵抗拜占庭节点，所以仅限于私链当中。因此如何将 Raft 算法进行改进，使其能够运用在联盟链，甚至在公链中是当下研究的热门话题。

将 Raft 算法移植到联盟链中最大的问题在于如何实现拜占庭容错[11]，拜占庭容错是要确保诚实的节点在受到恶意节点干扰的情况下也能达成共识，保证系统正常运行。PBFT 降低了拜占庭容错协议的运行复杂度，使算法复杂度从指数级别 $O(n(f+1))$降低到多项式级别 $O(n^2)$，使拜占庭协议在分布式系统应用中成为可能，也是现在主流的联盟链中所使用的共识算法。在 PBFT 中最大容错节点数量是$(n-1)/3$，而在 Raft 算法中最大的容错节点(宕机节点)数量是$(n-1)/2$，以及算法通信复杂度上 Raft 的 $O(n)$也是远少于 PBFT 的 $O(n^2)$，所以研究者们尝试改进 Raft，使之同时具有 Raft 和 PBFT 的优点。

2016 年，斯坦福的 Christopher Copeland 和 Hongxia Zhong 在论文《Tangaroa: a byzantine fault tolerant raft》[12]中提出在 Raft 算法基础上借鉴 PBFT 算法的一些特性(包括签

收稿日期：2022-03-22；**修回日期：**2022-04-29　**基金项目：**国家自然科学基金资助项目(61762046，62166019)；江西省教育厅科学技术研究项目(GJJ209412)；国家级大学生创新创业训练项目(201913434005)

作者简介：李淑芝(1964-)，女，江西赣州人，教授，硕士，主要研究方向为软件工程、信息隐藏；邹懿杰(1999-)，男，江西赣州人，硕士研究生，主要研究方向为区块链；邓小鸿(1982-)，男(通信作者)，湖北天门人，副教授，硕导，博士，主要研究方向为网络信息安全、区块链(deng_xh@jxust.edu.cn)；罗志琼(1999-)，女，江西吉安人，硕士研究生，主要研究方向为区块链；刘惠文(1987-)，女，江西吉安人，讲师，硕士，主要研究方向为区块链及其应用.

名、恶意领导探测、选举校验等)来实现拜占庭容错性，兼顾可实现性和鲁棒性。文献[13]中要求 Follower 节点在投出选票时在选票上进行签名，防止选票被伪造，其次要求客户端在发送日志时在上面签名，防止拜占庭的 Leader 伪造日志，在一定程度上提高了抗拜占庭节点能力。文献[14]针对 Raft 共识算法 Leader 节点选举中存在的多 Candidate 节点分票和 Follower 节点增多引发的投票效率问题，利用双层 Kademlia 协议建立的 K 桶实现 Candidate 节点集合内的稳定选举，并且针对 Raft 共识算法日志复制过程中，Leader 节点单节点日志复制过程效率低以及节点负载不均的问题，提出了均衡 Leader 节点负载的多 Candidate 节点并行日志复制方案。文献[15]中采用了两级共识机制，将 Raft 中的所有节点进行分组，每个组内选出领导者组成网络委员会，组内采用 Raft 共识，网络委员会中采用 PBFT 机制进行共识，是将 Raft 与 PBFT 结合的一种方式，但是无法有效抑制组内拜占庭节点的存在。文献[16]中将 Raft 选票拉取过程转换成阈值签名过程，阻止了拉取空白选票的行为，以及引入增量哈希保证日志不可窜改，增量哈希使得日志体增大，有一定的局限性，而且不能阻止拜占庭节点成为 Candidate 强行拉取选票将 Leader 进行更换。

综上所述，Raft 算法可以通过优化来抵御拜占庭节点，但仍需解决如下问题：a)日志易窜改，没有进行加密处理；b)选票的投出没有保障，仅仅靠任期大小判断是否将选票投出不够安全以及选票容易造假。针对上述问题，提出 RB-Raft(Resist Byzantine-Raf)算法，主要创新点如下：

a) 对于拜占庭的 Leader 节点容易伪造日志，造成日志被窜改，采用基于哈希链的动态日志验证机制，实现了日志防窜改以及日志可验证。

b) 对于选票造假以及选票容易被拜占庭节点利用来更换当前 Leader 的问题，设计一种基于门限加密的“遗书”机制，当 Leader 节点并未宕机时，Follower 节点都不会将选票投出，除非拿到 Leader 节点宕机后的遗书，而遗书的获取需要通过门限加密得到，避免了 Leader 节点被拜占庭节点异常替换。

1　问题的提出

1.1　Raft 工作流程

在 Raft 算法中有三种角色，每个节点都能担任不同的角色，但是不能多种角色存在于同一个节点上。这三种角色分别是:Leader(领导者)、Follower(跟随者)、Candidate(候选者)。每个角色都有着不同的任务：Leader 负责与客户端交互信息，并将日志信息同步给 Follower 节点，与 Follower 通过 HeartBeat 保持联系；Follower 负责响应 Leader 的日志同步请求，响应 Candidate 的投票选举请求，将 Leader 的日志文件同步到本地，在所有节点刚启动的时候都是为 Follower 状态；Candidate 负责选举投票，当 Leader 宕机时，通过投票选举转为 Leader 状态。Raft 算法开始时需要在集群中选举出 Leader 来负责日志复制的管理，Leader 接受来自客户端的事务请求(日志)，并将它们复制给集群的其他节点，然后负责通知集群中其他节点提交日志，Leader 负责保证其他节点与他的日志同步，当 Leader 宕机后集群其他节点会发起选举选出新的 Leader。Raft 算法工作流程基本由 Raft 节点状态机决定，如图 1 所示。

Raft 中采用心跳机制操控着 Leader 宕机后节点的重新选举。每个 Leader 都会向集群中的所有的节点发送心跳信号，证明着自己还存活未宕机。当 Leader 节点宕机时无法发送心跳信号，此时当 Follower 节点在规定的一段时间内没有收到来自 Leader 的心跳后会将自身的状态切换成为 Candidate，向其他节点发送选举请求，如果自身的日志比对方节点的日志更新以及 Term 更大就会收到对方节点的选票，否则对方节点将拒绝投票。倘若获取到的选票数量到达 $n/2$ (n 为节点个数)以上时，会将自身的状态置为 Leader，并任期号加 1。若在投票选举过程中重新收取的到 Leader 的心跳信号，就会将自身状态重新置为 Follower。

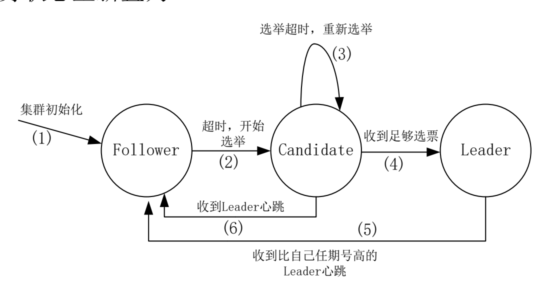


图 1　Raft 节点状态机

Fig. 1　Raft node state machine

1.2　选票造假问题

类似于 PBFT 中的视图,Raft 使用了一个 Term 的概念，当 Term 增加时相当于 PBFT 中的视图切换，每个 Term 都是一个连续递增的编号，Leader 身份由一个节点更换成另一个节点的时间就是一个 Term 周期，在一个 Term 中只能产生一个 Leader。Raft 算法开始时所有的 Follower 的 Term 为 1，其中一个 Follower 逻辑时钟到期后发现集群中没有 Leader 心跳，即转换为 Candidate，Term 加 1，此时任期为 2。谁的 Term 更大，谁就有更高的优先选择权力， Term 大的节点不会向 Term 小的节点投出选票，若 Leader 发现自身的 Term 小于某一个 Follower 节点，则该 Leader 会自动成为 Follower 节点。可以说每次 Term 的递增都将发生新一轮的选举，在 Raft 正常运转中所有节点的 Term 都是一致的，在节点不发生故障时一个 Term 会一直保持下去，当某节点收到的请求中 Term 比当前 Term 小时则拒绝该请求。

Raft 集群中的选票数量影响着谁能够成为 Leader，因为在原始的 Raft 中，谁的选票数量多，就代表着谁发现 Leader 节点宕机的时间越早，节点是无条件将选票投出给比自己 Term 大的节点，即成为 Candidate 节点的时间也越早，Term 比其他节点要更大，所以算法为了更快地选出新的 Leader，就选择以选票数量为作为挑选 Leader 的指标，使得集群能够快速恢复到正常状态。而在 Raft 算法中节点为何种身份由自身的状态机决定，即拜占庭节点可以随意更换自己身份，当本轮 Leader 节点没有宕机的时候，拜占庭节点可以成为 Candidate 节点，增大自身的 Term，向其他节点拉取选票，当获取到足够的选票以后就会将本轮的正常 Leader 节点替换。

1.3　日志易伪造问题

日志复制是 Raft 算法的核心之一，保证了 Raft 数据的一致性。在一个 Raft 集群中只有 Leader 节点才能接受客户端的请求，然后由 Leader 向其他 Follower 转发请求日志，Leader 不会删除任何日志，Follower 只会接收来自 Leader 所发送的日志信息。日志结构如图 2 所示，其中 x，y 代表的是某种指令，日志由序号跟条目(内容)组成，每个条目又由任期(Term)和指令(command)组成，committed 范围为超过半数以上的节点接受并储存的日志。

在 Raft 集群中可通过条目索引号、任期号来确定唯一的条目，并且该条目之前的所有条目都是一致的，当 Leader 节点宕机后，新的 Leader 被选举出来，此时集群中所有的节点会以新 Leader 的本地日志结构为标准进行同步，仅保留序号与任期号完全相同的部分，超出部分会被删除，少于部分会进行同步，使集群中日志保持一致。

从上述的 Raft 的日志复制过程以及日志结构可以看出，

Raft 中的 Follower 节点是无条件的接受并与 Leader 的日志结构保持相同的，即使自己的日志是比 Leader 节点更新。而日志消息又是由 Leader 节点进行打包并传播给其他的节点，若此时 Leader 节点为拜占庭节点，有可能将客户端传入的日志消息进行窜改。所以确保日志内容不被伪造，不被窜改对于抵抗拜占庭节点算法来说至关重要。

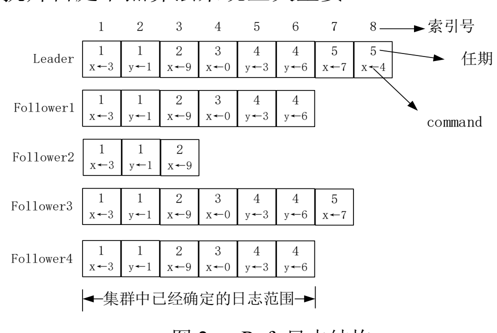


图 2　Raft 日志结构

Fig. 2　Raft Log structure

2　RB-Raft 算法

2.1　基于哈希链的动态日志验证机制

“哈希链”概念最早是由 Lamport 于 1981 年在《Password authentication with insecure communication》[17]一文中提出，旨在解决密码在传输过程中会被入侵者窃取、窜改的问题。哈希链通过哈希函数对密码进行多次迭代加密，具有良好的抗干扰性，且服务器端只需保存最后一次加密的密文即可验证全部密文序列。因为日志结构要保证日志顺序不变，每块日志体不变，因此采用哈希链的方式将所有的日志块串联起来，只需要验证最末尾哈希值即可验证前面所有的日志块是否相同。

1.日志哈希链的生成

当客户端在批量打包日志的时候，对日志块进行哈希链运算，日志哈希链生成过程如图 3 所示。

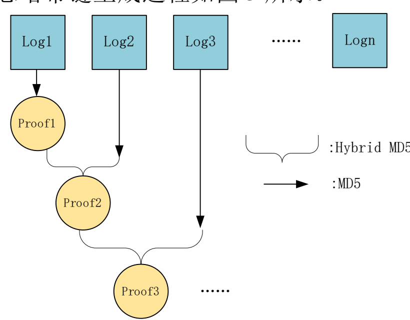


图 3　日志哈希链生成过程

Fig. 3　Log hash chain generation process

其中，Log 代表日志块体，Proof 代表验证哈希，算法描述如下：

算法 1　构造日志哈希链

```
输入：日志块序列 Log[]
输出：日志哈希链表 ProofTable
Int i,j=1//定义 i 为日志循环编号，j 为 proof 循环编号
while(i!=MaxLength(Log))//没有将哈希链覆盖到全日志时
  string file=getLogBlockToflie(Log[i])//获取日志
  int proof//每次循环定义一个新的 proof
  if i==1 //判断是否为第一块日志
    proof=MD5toFile(file);
    ProofTable[j]=proof;
    j++,i++;
    break;//进入下一个循环
  end if
  proof=MD5toFile(file);
  ProofTable[i]=HybridMD5(proof,ProofTable[i-1]);//混合哈希
  j++,i++;
  break;
end while
```

以上为哈希链生成部分全过程，接着需要对 Follower 节点拿到的日志块进行校验。在本文中采用 Follower 节点向客户端发送校验信息的方式。

当需要对日志进行校验时，Follower 节点将自身所存储的日志做哈希链计算得出最后一块日志块的 Proof，此 Proof 代表了本节点所有的日志块，然后将最后一块日志块序号作为请求参数发送给客户端，客户端返回相应的 Proof，节点与其进行对比，假如对比失败，两者完全不相同，说明自身日志与客户端发送过的日志存在不一致。

当日志对比失败后首先需要对日志进行回滚，回滚到上一层 Proof 再次进行对比，不一致的话就继续回滚，直到 Proof 一致则不再进行回滚，错误 Proof 的日志块将会被删除，并且向客户端获取同步错误的日志块。

2.日志动态验证机制

日志对比失败存在几种可能性：1、当前 Leader 节点为拜占庭节点，对日志进行了修改。2、之前 Leader 节点存在拜占庭节点，只不过是没有被发现。3、在传输过程中因为一些异常情况如：网络波动，I/O 读写错误，导致日志缺少，错误。因为无法确定是何种原因导致日志块发生错误，所以当日志发生错误时不能直接否定该 Leader 节点。因此设计以下动态验证机制用来对上述三种情况进行容错：

每个 Follower 节点会每隔 T 份日志向客户端发送检测请求，返回当前本地日志索引号所对应的 Proof，并将本地日志做迭代式哈希，判断是否一致。本文利用对数函数在 1 附近能够快速收敛的性质，设计了如式(1)所示分段函数：T_n 代表第 n 轮 T 值，x 的作用类似于信用值，初始值为 1，步长为 0.1。

$$T_n=\begin{cases}T_{n-1}*2^{\lceil Log_2^x\rceil} & \text{err}\neq null\wedge x\leq 1\\ T_{n-1}*2^{\lfloor Log_2^x\rfloor} & \text{err}\equiv null\wedge x>1\end{cases}\tag{1}$$

当验证错误时，应该缩短检测的份数，即增大检测频率，如 x 初始值为 1，当验证发现错误时，x 首先会降低 0.1，当 x 小于 1 时 Log_2^x 为负数，并且向上取整，此时 2^x 为减函数，导致 T_n 小于 T_{n-1}，缩短检测跨度，加快了检测频率。而当检测正确时，x 增加 0.1，当 x 大于 1 时 Log_2^x 为正数并且向下取整，此时 2^x 为增函数，导致 T_n 大于 T_{n-1}，使得检测跨度变大，检测频率降低，以避免增大客户端的检测压力，符合本算法的需求。

当 T 变成 1 时，即 Leader 节点每同步一次日志，Follower 节点就需要向客户端进行检验，说明该 Leader 节点在该 Follower 节点中已经不可信，当这种 Follower 节点达到一半时，客户端会将其替换掉，重新再竞选一个新 Leader 节点。何时替换 Leader 节点的主要由三个因素影响：T 值大小，Leader 节点发送错误日志份数，以及集群同步速度，后面两个因素都无法人为的控制，但是可以通过控制 T 值的初始大小来设置该机制的严厉程度，T 值的下降速度为 Log_2^T，当 T 较大时，就允许较大的错误率，当 T 较小时就是相对比较严格。

2.2　基于门限加密的“遗书”机制

“遗书”最早理解为通过公开的信件来约定好如何分配已去世人财产等身后事宜。在 Raft 算法中当 Leader 节点宕机后，Follower 节点仅仅依据任期大小来选择是否将选票投出，而任期的大小又是通过是否接收到来自 Leader 节点的心跳信号而决定的，这是很容易被拜占庭节点所利用，拜占庭节点可以通过修改自身任期，来达到获取大量选票从而替换当前正常的 Leader 节点。

为了解决上述问题，本文设计“遗书”机制，在节点成为 Leader 时产生一份遗书(立下遗嘱)，其他节点获取到遗书内容才有资格向 Follower 节点拉取选票(分配财产)，而遗书内容只能当 Leader 节点宕机后才可以打开。为了确保“遗书”的安全性，本文利用门限加密对“遗书”进行加密，门限加密由 Desmedt 和 Frankel 等人[18]提出的。门限密码通过将密钥信息发放给多个用户进行存储，任意少于门限数量的密钥信息是无法进行解密，必须拥有超过门限数量的密钥信息才能获取到密文信息。只有当大部分的 Follower 节点将门限密钥给出才能够将遗书进行解密(打开)，从而获取到遗书内容并拉取选票(分配财产)，而是否给出门限密钥又取决于 Leader 节点的心跳信号(该过程类似于验证 Leader 节点是否存活)，形成了一个完整的验证闭环，避免了拜占庭节点修改任期将正常的 Leader 节点异常替换，具体步骤如下：

在基于门限加密的“遗书”机制中，将“遗书内容”定义为需要加密的密文(*Lmsg*, Legacy-message)，设定 Leader 编号以及成为 Leader 时间为原始数据即 *Lmsg*。

遗书生成阶段：Leader 节点首先使用 $KeyGen(\lambda,n,t)$ 生成密钥，输入安全参数 λ，用户数量 n 和门限值 t，门限值一般设定为集群数量的一半，输出公钥 *pk* 和节点私钥分享 $sk=(skid_1,\cdots,skid_n)$，通过 $Enc(pk,Lmsg)$ 获得门限加密后的密文 L。

分发密钥及遗书阶段：向所有节点发送一份“遗书”(L，即对 *Lmsg* 门限加密以后的密文)、该 Follower 节点所对应的私钥分享 sk_{idj} 以及该 *Lmsg* 的散列值 *HashLegacy*。

遗书解密阶段：当 Follower 节点在心跳超时器结束以后依旧没有收到心跳，那么该节点会通过向其他节点获取解密分享 L_{id} 的方式对 L 进行门限解密，解密算法为 $Dec(sk_{id},L)$，输入自身私钥分享 sk_{id} 以及 L，得出 L_{id}。此时该节点必须拥有 t 个 L_{id} 通过 $Combine(Lid_1,\ldots,Lid_t)$ 算法才能获取到 *Lmsg*。其中当节点收到来自其他节点的门限请求时，会判断是否距离上次收到心跳信息是否超过 M 时间，超过 M 时间则将门限私钥给出，否则拒绝交出门限私钥。当 Leader 节点并未宕机时，必定有节点收到心跳在 M 时间内，所以就推翻了 Leader 节点宕机的这一定论，因此不会将私钥分享出去。M 值的设置为 Follower 节点心跳超时器时间的 2/3,在实验中发现大部分节点都在前 2/3 的心跳超时时间中收到心跳信息，几乎没有超过该长度的节点，所以将 M 设置为该值，便于加快效率。

选票拉取阶段：当获取到遗书内容后会将自身状态转换成为 Candidate 向其他 Follower 节点发送 RequestVote 消息拉去选票，而被拉取选票节点需判断来自 Candidate 节点的 *Lmsg* 是否正确，通过对 *Lmsg* 进行哈希运算与之前 Leader 节点发送的 *HashLegacy* 进行对比，对比通过才允许进入正常的投票判断环节，否则不对其进行投票操作。

遗书生成和解密大致流程如图 4 所示，之后的正常选票拉取阶段在 2.1 节中已经介绍，不再赘述。

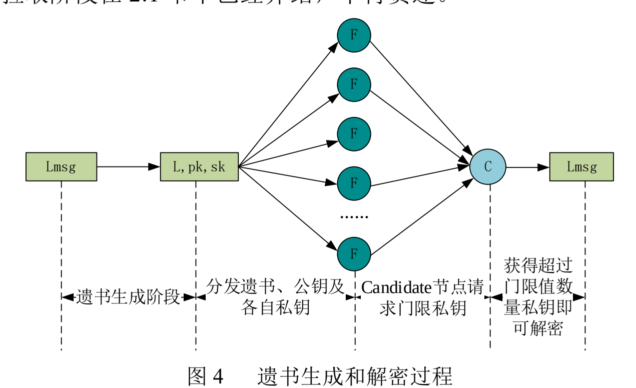


图 4　遗书生成和解密过程

Fig. 4　Suicide note generation and decryption process

基于门限加密的“遗书”机制部分算法描述如下：

算法 2　遗书生成算法

```
输入：Lmsg,n,t,λ。
输出：sk,pk,L。
if node.status=Leader and flag=true then //节点成为Leader
 pk,sk = KeyGen(λ,n,t) //公私密钥生成
 L=Enc(pk,Lmsg) //遗书加密
 HashLeagcy=h(Lmsg)
 Broadcast(pk,sk,L, HashLeagcy) //广播遗书
end if
```

算法 3　门限分享算法

```
输入:L,skid 。
输出：Lid 。
if node.status=Follower and getLMessage then //Follower节点被请求门限私钥
if T>timeout then //当T值大于超时时间
   return Lid=Dec(skid,L) //给出节点门限私钥
else
return err
```

算法 4　遗书解密算法

```
输入：Lid1,⋯,Lidj,L 。
输出：Lmsg 。
 if  node.status=Follower  and  heartbeat=false  then //Follower节点未收到心跳信息
 if count(Lid1,⋯,Lidj)>=t then
   Lmsg=Combine(Lid1,⋯,Lidj) //遗书解密
   node.status=Candidate
   return Lmsg
 else
 return err
```

算法 5　遗书验证算法

```
输入：Lmsg, HashLeagcy 。
输出：vote 。
if node.status=Follower and getLmsg then //收到其他节点的投票请求
 if HashLeagcy==h(Lmsg) then
   进入正常投票判断
   if 通过判断 then
      return vote
 return err
```

3　安全性分析

RB-Raft 算法为能够抵抗拜占庭节点攻击的 Raft 算法，在安全性方面需要满足以下几个要求：1、日志不允许被 Leader 节点修改，即所有节点提交的日志需要保证一致，这是 Raft 作为一种共识算法所需要具备的最重要的特质。2、确保 Leader 不被异常替换，即被恶意的 Candidate 拉取选票导致 Term 切换。

3.1　日志完整性

要保证日志完整性就要确保两点：1、日志顺序不变。2、每个日志块是相同的。以下是日志完整性分析：

定义 1　记日志块为 B_i，i>0 且 i 为正整数，哈希序列为 H_j，j>0 且 j 为正整数，i、j∈R。

定义 2　h(Text)为哈希函数，在本文中采用 MD5，返回哈希值。

由上述算法流程可知：

$$H_j = h(h(h(h(B_1),B_2),\ldots),B_{i-1}),\ j=i$$

恶意的拜占庭节点将其中任意一个日志块 Bk 修改部分

内容后得到的日志块记为 B'_k，得到 H'_j 如下:

$$H'_j = h(h(h(h(h(h(B_1), B_2), \ldots), B'_k), \ldots), B_i), i = j > k$$

因此必然存在 ,即可判定在日志序列中某份日志被窜改导致 H_j 改变，发现被窜改以后可以向客户端同步被窜改的日志，保证了日志序列的完整性和真实性。

3.2　节点选举安全性

在 Raft 算法中，Leader 节点需要定时向 Follower 节点发送心跳信息 AppendEntries，用来向其他节点宣布自己仍然存活，可以正常工作，当 Leader 节点宕机时，因 Follower 节点在自身设置的心跳超时时间内还未收到来自 Leader 节点的 AppendEntries 消息，所以会将 Term 增加，并且转换成为 Candidate 身份进行选举，向其他 Follower 节点拉取选票成为新的 Leader 节点。

而在存在拜占庭节点的环境下，Candidate 拉取选票的时间缺乏限制，即使 Leader 并没有宕机，恶意的 Candidate 节点可以通过拉取选票将当前 Leader 替换(因为 Candidate 的 Term 比 Leader 大，所以会将手中的选票投出)，为避免这种情况的发生，提出了基于门限签名的“遗书机制”来避免 Leader 节点的异常更换，通过门限签名方式保证了拜占庭节点的数量必须超过门限值 t 才会获取到遗书，通常 t 值一般会设置为 n/2，具体方式详见 3.2 章节。该方式确保 Candidate 节点的拉取选票是有证据(未收到 Leader 心跳)的背书，保证了节点选举的安全性。

3.3　抗拜占庭节点能力分析

Raft 算法本身是属于私链的一种共识算法，是不允许拜占庭节点的存在，只允许部分宕机节点的存在。而 RB-Raft 是以 Raft 为原型改进的联盟链算法，允许拜占庭节点的存在，所以主要与联盟链中主流的 PBFT 算法进行对比。

相对于联盟链来说，能抵抗攻击的指标主要取决于能够允许多少拜占庭节点的存在。假设集群节点总数为 n，拜占庭节点个数为 f。在 RB-Raft 中，根据少数服从多数原则，集群中的正常节点只需要比 f 个节点再多一个节点即可进行正常的共识，可得 $n=2f+1$，因此 RB-Raft 算法支持的最大容错节点数量 f 为 $(n-1)/2$。在 PBFT 算法中，假设存在故障节点和拜占庭节点都是不同的节点，那么就会有 f 个故障节点和 f 个拜占庭节点，当发现故障节点后,算法会将其排除在集群外，因此正常节点需要比拜占庭节点多一个节点，可得 $n=3f+1$，因此 PBFT 算法支持的最大容错节点数量 f 为 $(n-1)/3$。综上所述，在联盟链共识算法当中，RB-Raft 算法要比 PBFT 算法多允许 $(n-1)/6$ 的拜占庭节点的存在，抗拜占庭节点的能力比 PBFT 要高。

4　仿真实验

本文采用 go 语言实现了 RB-Raft 算法，然后采用本地单机多节点模拟共识过程，记录数据吞吐量、共识时延、以及拜占庭节点行为等数据，完成对比实验。最后通过实验对比表明该算法具有抗拜占庭能力以及高吞吐量、低延时优点。

4.1　抗拜占庭节点性能测试

在本节实验中，引入可控的拜占庭节点以用来测试集群抗拜占庭能力，并且采取不同身份以及行为进行实验，具体实验过程如下：

1.日志防伪测试

在本小节中首先对日志防伪性能进行测试。由于选举过程的随机性，无法指定可控节点作为 Leader，所以将可控节点的选举超时器的时间设置为远小于其他正常节点，而又比正常通信所花费时间长，使得集群初始化时可控节点能够第一时间进行选举，从而成为 Leader，便于实验的开展。然后从客户端发送相同的 500 份日志到 Leader 节点中，Leader 节点采用伪随机数生成[1,500]的 10 份日志序号，并进行修改，监测 Follower 节点是否能够正确发现被窜改日志并回滚。上述操作分别在 Raft 与 RB-Raft 集群中开展，得到实验结果如表 1 所示。

表 1　日志防伪测试结果

Tab. 1　Log security test result

算法	窜改日志序号	发现被窜改日志序号	准确率
Raft	383,11,484,103,133,211,371,368,75,320	无 无 无 无 无 无 无 无 无 无	0%
RB-Raft	327,33,451,440,30,179,98,117,254,413	327,33,451,440,30,179,98,117,254,413	100%

表中可以看出 Raft 算法与 RB-Raft 算法在准确率上面属于两个极端，Raft 算法完全无法发现错误日志，对任何 Leader 节点发送过来的日志都进行了同步，显然在存在拜占庭节点的情况下是不可取的，而 RB-Raft 算法对随机窜改的日志块都能够准确的发现，并且进行回滚，准确率到达了 100%，因此 RB-Raft 具有良好的抵抗日志伪造的优点。

接着需要对动态验证机制进行实验，测试对于恶意的 Leader 节点能否快速将其替换，对于正常的 Leader 节点能否缩短检测频率以减轻节点验证压力。在该实验中，以 Follower 节点的视角，分别测试拜占庭 Leader 节点修改日志和节点正常的两种情况中 Follower 节点的 T 值变化曲线，实验中 T 值初始值都设置为 8，实验结果如图 5 所示。

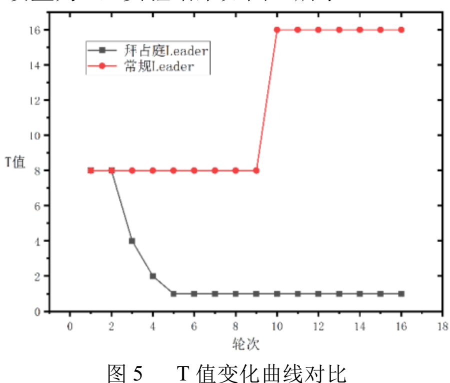


图 5　T 值变化曲线对比

Fig. 5　T value change curve comparison diagram

图中可以看出常规 Leader 节点在第 11 轮次后将 T 值增大到了 16，增大了检测跨度，降低了检测频率，减轻了节点压力。而拜占庭 Leader 在第二轮次修改日志后 T 值快速降低到 1，即达到了不可信的状态，使得恶意的 Leader 节点被替换，达到了抵抗拜占庭节点的攻击目的。上述实验主要是由于式(1)的分段函数所致，x 值可以类比于信用值，通过 x 的增减来影响 T 值的变化。

2.遗书机制测试

在本节实验中同样需要引入可控的恶意节点，引入方法与上节相同。然后通过可控节点在 Leader 节点并未宕机的情况下分别向 Raft 算法集群和 RB-Raft 算法集群中节点发出选票拉取，在 20ms 后统计两个集群中收到的 Follower 节点的选票数量，实验结果如表 2、3 所示。

表 2　100 节点数量遗书机制测试指标对比

Tab. 2　100 number of nodes Suicide note mechanism test index comparison

指标	Raft	RB-Raft
收到选票数量	87	0
门限签名数量	-	15
是否获取到遗书	-	否

从上表中可以看出，对于 Raft 算法中节点的拜占庭行为是无法遏制住的，在 Leader 节点未宕机的情况下依旧获取到了半数以上的节点选票，因此可以将正常的 Leader 节点替换。而在 RB-Raft 中获取到的选票数量都为 0，门限签名数量也分别为 15%、14.4%，不足门限阈值，因此获取不到遗书内容，自然无法获取到其他节点的选票，避免了节点恶意将

Leader 节点替换。

表 3　500 节点数量遗书机制测试指标对比

Tab. 3　500 number of nodes Suicide note mechanism test index comparison

指标	Raft	RB-Raft
收到选票数量	416	0
门限签名数量	-	72
是否获取到遗书	-	否

4.2　共识时延

共识时延是衡量共识算法的一项重要指标，在共识算法中表示从客户端发出命令开始到客户端收到集群发出同步完成命令的时间差。在本节中主要对经典 PBFT 共识机制，Raft 算法，以及 RB-Raft 算法的共识时延进行对比，如图 6 所示。

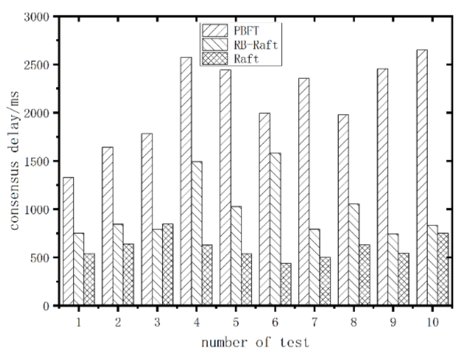


图 6　共识时延对比

Fig. 6　Consensus delay comparison diagram

实验结果显示 PBFT 的时延大部分比 Raft 跟 RB-Raft 算法要更高的，其原因还是在于算法的复杂程度上，PBFT 要经过多阶段的通信才能达到集群一致。而 RB-Raft 算法因为增加的密钥发放以及遗书机制，时延方面要比 Raft 稍高，但也是在可接受范围之内。因此 RB-Raft 算法损失了一定的效率换取到了抵抗拜占庭节点的能力。

4.3　吞吐量测试

数据吞吐量是衡量区块链性能的一种重要工具，代表单位时间内处理的交易数量，表示为 TPS(Transactions Per Second)。共识效率是影响 TPS 的主要因素，共识效率越高，事务处理能力就越强。影响 TPS 的因素还包括节点对并发数据的处理能力，对数据库的 I/O 读写能力等，测试过程采用 EOSBenchTool 工具。在本节实验中采用控制变量法，只将算法作为唯一变量，其他环境因素均不变，对比了 Raft、PBFT、以及 RB-Raft 算法吞吐量，最后随机选取测试数据中的 20 组进行比较，得出吞吐量测试结果如图 7 所示。

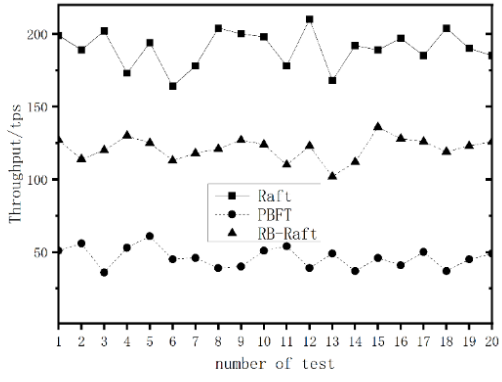


图 7　吞吐量测试对比

Fig. 7　Throughput test comparison diagram

实验结果显示，RB-Raft 算法与 Raft 算法都具有高数据吞吐量，吞吐量比 Raft 算法低了 39.1%，但是比 PBFT 算法提高了 61.8%，因为实现了拜占庭容错，节点交互时会互相验证身份，以及集群的数据通信量增加，所以吞吐量会相对于 Raft 下降，但是相比于 PBFT 的数据吞吐量还是较高的。

4.4　算法性能对比

本小节对 PBFT 算法、Raft 算法、RB-Raft 算法部分性能进行汇总比较，比较结果如表 4：

表 4　算法性能对比

Tab. 4　Algorithm performance comparison

算法	允许存在的拜占庭节点数量(共n 个节点)	平均共识时延/ms	平均吞吐量/ps	算法通信开销
PBFT	$(n\text{-}1)/3$	2119.8	46.25	$O(n^2)$
Raft	0	604.5	189.95	$2(n\text{-}1)$
RB-Raft	$(n\text{-}1)/2$	989.9	121.2	$3(n\text{-}1)$

表中可以看出在抗拜占庭节点方面，因为 PBFT 算法容错 f 个节点需要 $3f$+1 个总节点，所以其允许存在的拜占庭节点数量为$(n\text{-}1)/3$。而 Raft 是不允许拜占庭节点存在的，但是 RB-Raft 算法只需要保证有半数以上的节点能够成功同步日志即可，所以是允许$(n\text{-}1)/2$ 的拜占庭节点存在，所以 RB-Raft 算法是允许存在拜占庭节点最多的，即其抗拜占庭节点能力是最强。而在共识时延方面，RB-Raft 算法是小于 PBFT 但是比原始 Raft 算法还是稍微要大一些，在吞吐量方面也同样是要大于 PBFT 但是要小于原始 Raft 算法，而通信开销方面是远小于 PBFT 的 $O(n^2)$，与原始 Raft 算法相当。

接着将本文算法与其他 Raft 拜占庭容错类进行比较，结果如表 5。

表 5　Raft 拜占庭容错类比较

Tab. 5　Raft Byzantine fault tolerant class comparison

算法	允许存在的拜占庭节点数量(共 n 个节点)	Leader 选举时间	是否日志加密	是否选票验证
RB-Raft	(n-1)/2	与 Raft 相当	是	是
文献[13]	(n-1)/2	与 Raft 相当	是	仅选票签名
文献[14]	0	远低于 Raft	仅优化	否
文献[15]	4f1f2+2f2+f13(m>2f1+1,k>3f2+1,m 为组内个数，k 为分组数，k*m=n)	与 Raft 相当	否	否

从表中可以看出本文算法允许存在的拜占庭节点与文献[13]相当，比其他算法要更高，但是在 Leader 选举时间方面并没有优化。文献[14]利用双层 Kademlia 协议建立的 K 桶实现 Candidate 节点集合内的稳定选举，因此文献[14]是这些文献中 Leader 选举时间最短的算法。其中文献[15]是一种分组思想，采用了组内 Raft 组间 PBFT 的共识机制，决定容错率的还是由 PBFT 算法所决定，所以其对 Raft 算法没有比较大的改进。但是本文提出的 RB-Raft 算法在日志加密、选票验证方面进行了优化，在抗拜占庭节点的全面性上做了较好的工作。

5　结束语

本文提出了一种可抵抗拜占庭节点的 Raft 算法——RB-Raft 算法，解决了 Raft 在不确定的环境下的日志篡改和不能拜占庭容错的问题。首先采用哈希链的方式对日志进行处理，并且通过动态验证机制进行日志验证，解决了日志伪造问题并降低节点的验证压力。其次，提出基于门限加密的“遗书”机制，通过门限加密使得接收心跳信号拥有节点的肯定作为背书用于防止拜占庭节点拉取选票更换 Leader 节点，解决了拜占庭节点影响系统一致性的问题。最后，实验数据表明：RB-Raft 算法相比于 Raft 算法具有抗拜占庭节点的能力，其算法吞吐量比 PBFT 提高了 61.8%，平均共识时延比 PBFT 降低了 53.3%。综上所述，RB-Raft 适合联盟链在不安全的环境下进行共识，如车联网，物联网这种要求共识效率高，而又容易存在恶意节点对系统攻击的应用需求。RB-Raft 算法具有共识低时延和高安全性，适用于在车联网等去中心化环境下进行共识，未来的工作将围绕着如何把本算法应用到车联网、物联网场景当中。

参考文献：

[1] Nakamoto S. Bitcoin: A peer to peer electronic cash system [EB/OL]. 2008. https://bitcoin. org/bitcoin. pdf.

[2] Ferretti S, D'Angelo G. On the ethereum Blockchain structure: A complex networks theory perspective [J]. Concurrency and Computation: Practice and Experience, 2020, 32 (12): e5493.

[3] Wang G, Zhang S, Yu T, *et al.* A systematic overview of Blockchain research [J]. Journal of Systems Science and Information, 2021, 9 (3): 205-238.

[4] Li W, He M, Haiquan S. An overview of Blockchain technology: applications, challenges and future trends [C]// 2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC) . IEEE, 2021: 31-39.

[5] Arnold R, Longley D. continuity: a deterministic byzantine fault tolerant asynchronous consensus algorithm [J]. Computer Networks, 2021, 199 (11): 108431-108443.

[6] 邓小鸿, 王智强, 李娟, 等. 主流区块链共识算法对比研究 [J]. 计算机应用研究, 2022, 39 (1): 1-8. (Deng Xiaohong, Wang Zhiqiang, Li Juan, *et al.* Comparative research on mainstream Blockchain consensus algorithms [J]. Application Research of Computers, 2022, 39 (1): 1-8.)

[7] Meneghetti A, Sala M, Taufer D. A survey on pow-based consensus [J]. Annals of Emerging Technologies in Computing (AETiC) , Print ISSN, 2020, 4 (1): 8-18.

[8] Li Y, Wang Z, Fan J, *et al.* An extensible consensus algorithm based on PBFT [C]// 2019 International conference on cyber-enabled distributed computing and knowledge discovery (CyberC) . IEEE, 2019: 17-23.

[9] Ongaro D, OusterhoutJ. In search of an understandable consensus algorithm [C]// 2014{USENIX}Annual Technical Conference ({USENIX}{ATC}14) . 2014: 305-319.

[10] Lamport L. Fast paxos [J]. Distributed computing, 2006, 19 (2): 79-103.

[11] Lamport L, Shostak R, Pease M. The byzantine generals problem [M]// Concurrency: the Works of Leslie Lamport. 2019: 203-226.

[12] Copeland C, Zhong H. Tangaroa: a byzantine fault tolerant raft [J]. Stanford University. 2016.

[13] Tian S, Liu Y, Zhang Y, *et al.* A Byzantine Fault-Tolerant Raft algorithm combined with schnorr signature [C]// 2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM) . IEEE, 2021: 1-5.

[14] 王日宏, 周航, 徐泉清, 等. 用于联盟链的非拜占庭容错共识算法 [J]. 计算机科学, 2021, 48 (09): 317-323. (Wang Rihong, Zhou Hang, Xu Quanqing, *et al.* Non-byzantine tault tolerance consensus algorithm for consortium Blockchain [J]. Computer Science, 2021, 48 (09): 317-323.)

[15] 黄冬艳, 李浪, 陈斌, 等. RBFT: 基于 Raft 集群的拜占庭容错共识机制 [J]. 通信学报, 2021, 42 (03): 209-219. (Huang Dongyan, Li Lang, Chen Bin, *et al.* RBFT: a new Byzantine fault-tolerant consensus mechanism based on Raft cluster [J]. Journal on Communications, 2021, 42 (03): 209-219.)

[16] 王日宏, 张立锋, 周航, 等. 一种结合 BLS 签名的可拜占庭容错 Raft 算法 [J]. 应用科学学报, 2020, 38 (01): 93-104. (Wang Rihong, Zhang Lifeng, Zhou Hang, *et al.* A byzantine fault tolerance raft algorithm combines with BLS signature [J]. JOURNAL OF APPLIED SCIENCES—Electronics and Information Engineering, 2020, 38 (01): 93-104.)

[17] Lamport L. Password authentication with insecure communication [J]. Communications of the ACM, 1981, 24 (11): 770-772.

[18] DESMEDT Y, FRANKEL Y. Threshold cryptosystems [C]. In: Advances in Cryptology—CRYPTO’89. Springer Berlin Heidelberg, 1989: 307–315. [DOI: 10. 1007/0-387-34805-0_28]